TYPE HIERARCHIES AND SEMANTIC DATA MODELS

Antonio Albano (*)
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

Abstract

The basic abstraction mechanisms of Semantic Data Models - aggregation, classification and generalization - are considered the essential features to overcome the limitations of traditional data models in terms of semantic expressiveness. An important issue in database programming language design is which features should a programming language have to support the abstraction mechanisms of Semantic Data Models. This paper shows that when using a strongly typed programming language, that language should support the notion of type hierarchies to achieve a full integration of Semantic Data Models abstraction mechanisms within the language's type system. The solution is presented using the language Galileo, a strongly typed, interactive programming language specifically designed for database applications.

1. INTRODUCTION

In the past the fields of programming languages and database languages have developed separately because each focused on different classes of problems. Research in programming language design has concentrated on creation of features to support the implementation of complex algorithms using temporary data (Shaw 80). Research in database language design, instead, has been mainly concerned with features to model persistent, interrelated data which must be accessed by programs or interactive query languages. An important exception to these divergent trends has been the common attempt by both programming and database researcher workers to design a basic set of abstraction mechanisms for data modeling. Still, the solutions provided have been quite distinct (Biller 78, Brodie 80,81; Schmidt 78; Weber 78).

Recently, this situation has been changing, largely because database people are paying more attention to the design of languages that besides types, abstract types and modularization, include abstraction mechanisms to support database models. For instance, proposals and implementations have been given to integrate a relational data model into a general-purpose, Pascal-like programming language (ASTRAL (Amble 79), PASCAL-R (Schmidt 80), PLAIN (Wasserman 79), RIGEL (Rowe 79), THESEUS (Shopiro 79)).

Another database goal, which will have far reaching impact on programming languages, is the design of a language for database applications which supports the basic features of Semantic Data Models. A Semantic Data Model is a set of data abstraction mechanisms to describe the structure of databases: the structures, and the associated operations, are explicitly intended to represent certain types of real-world information. A survey and an analysis of the motivations for this new generation of data models is reported in (McLeod 82). It is sufficient here to remember that the basic abstraction mechanisms are classification, aggregation and generalization. For the purposes of this paper, we here interested in considering the third mechanism, named also **IS-A** hierarchy, and originally proposed in the context of Semantic Networks.

(*) This work was supported in part by the Consiglio Nazionale delle Ricerche, Progetto Finalizzato Informatica, Obiettivo DATAID, and in part by Ministero della Pubblica Istruzione.
Present address: Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56100 Pisa, Italy.

The **IS-A** hierarchy is used in Semantic Data Models as a definitional mechanism involving two different notions (Wong 77). First, supposing Students **IS-A** Persons, it establishes an existence constraint among the elemens of Students and Persons present in the database: The elements of Students are in every state a subset of the elements of Persons (Extensional **IS-A** Constraints). Secondly, it is a compatibility rule between the elements of Students and Persons, in that every element of Students inherits all the properties of Persons elements (Structural **IS-A** Constraint). The inheritance rule, with this interpretation of the **IS-A** mechanism, is therefore **strict** rather than **default** (e.g. (Carbonell 81)). Consequently, elements of Students can be used in any context were an element of Persons is expected, by not vice versa (the compatibility rule is a partial order).

The question is which features should a programming language have to support this abstraction mechanism. If we think, for simplicity, of Students and Persons as identifiers bound to collections of values of type Student and Person, the first notion behind the **IS-A** hierarchies is a constraint on the values of the identifiers, while the second is a compatibility rule between the types of the elements.

Three of the database programming languages that have been proposed to address this problem are TAXIS, ADAPLEX and DIAL. TAXIS, which has the merit of being the first proposed, uses an approach not based on a typed programming languages (Mylopoulos 80). More closely related to the author's work is ADAPLEX, since the solution proposed is given within the framework of a strongly typed programming language, in this case ADA (Smith 81, Wegner 80). We consider this approach more interesting because we believe that the well known benefits of static typechecking are notable for database applications: The task of modeling becomes easier and more productive (Brodie 80, Biller 78). The solution adopted by ADAPLEX, however, is ad hoc for modeling databases, and it is not an independent feature of the language that can be used also for modeling temporary data. Similar considerations apply to DIAL (Hammer 80), which has evolved from SDM (Hammer 81): It is a programming language with data types, but the features for database medeling are not integrated with the data type system. We claim that a better solution could be achieved if the type system of the language would support the notion of type hierarchies.
We will discuss the solution adopted in Galileo, a strongly typed, interactive programming language, which integrates Semantic Data Model abstraction mechanisms into the framework of the language Edinburgh-ML (Gordon 79b, Albano 82). In particular, Galileo provides two independent features: 1) a type system with type hierarchies, and 2) the Class mechanism to deal with databases. When these features are combined in defining derived classes, an **IS-A** hierarchy is modeled.

A complete description of Galileo is outside the scope of this paper; it has been given in (Albano 82) and, together with the denotational semantics, in (Capaccioli 83). In the next section we give an overview of the language. Section 3 describes the notion of type hierarchies, and in Section 4 we present the class mechanism to deal with databases and to model the **IS-A** hierarchies.

2. OVERVIEW OF Galileo

Galileo is not a Semantic Data Model, but it is a strongly typed programming language which supports the following abstraction mechanisms of Semantic Data Models to design a database application:

Classification: Entities of the world being modeled that share common characteristics are described by the type of the elements of a class. The name of the class denotes the elements currently present in the database. The elements of a class are represented uniquely; no copies of them are allowed.

Aggregation: Elements of classes are aggregates, i.e. they are abstractions of heterogeneous components and may have elements of other classes as components. Associations among entities are represented by aggregations in a Galileo database. Components of elements of classes can be collections of homogeneous values to represent, for example, multivalued associations among entities. Moreover, because of the unique representation of elements of classes, any modification of an element is reflected anywhere that element appears as component.

Generalization: Elements of a class can be described in different ways by means of derived classes. Elements of a derived class also belong to the parent class from which the class is derived using a predefined set of operators. The derived classes mechanism includes the **IS-A** hierarchy of Semantic Networks and Semantic Data Models.

Modularization: Data and operations can be partitioned into interrelated modules. Therefore, a complex schema can be structured into smaller, meaningful and manageable units. For instance, a unit may model a user view or a description of the schema produced by a stepwise refinement methodology by specialization.

Other features of Galileo are:

1. It is an expression oriented language, in that

each construct is applied to values to return a value. This feature is interesting because it allows the interactive use of Galileo without resorting to a new, stand-alone query language.

2. It is higher order, in that functions are denotable values of the language. Therefore, a function can be a component of an aggregate which represents an entity, e.g. an age may be described as a function of the birthdate.

3. Every denotable value of the language possesses a type:

   a. A type is a set of values sharing common characteristics, together with the primitive operators which can be applied to these values.

   b. The predefined types of the language are **bool**, **num**, **int**, **string**, equipped with the usual operators, and the type **null**, which is a singleton set with the element nil, equipped with the equality operator.

   c. The type constructors available to define new type names, from predefined or previously defined types, are: Tuple (record), sequence, discriminated union (variant), function, modifiable value (reference), and abstract types. There are two constructors for abstract types: ⇔ and ↔ . The former is similar to CLU clusters (Liskov 77), ALPHARD forms (Shaw 77, 81) or Euclid modules (Lampson 77). It is used to define a new type together with the operations available. The latter is similar to the type constructor of Ada: it defines a new type which inherits the primitive operations of the representation type.

   d. The type system supports the notion of type hierarchy, in that if a type t is a subtype of a type t', then a value of t can be used as argument of any operation defined for values of t', but not vice versa because the subtype relation is a partial order. The type hierarchy is a directed acyclic graph instead of a simple tree.

4. Every Galileo expression has a type. The meaning of "an expression e having type t" is that the value of e possesses the type t. In general, any expression has a type that can be statically determined, so that every type violation can be detected by textual inspection (static type checking). However, if the type checker is not able to ascribe a type to an expression, the user must specify the type with the notation "Expression: Type". The language has been designed to be statically type checkable for two reasons: First, for the considerable benefits in testing and debugging; secondly, because programs are safely executed disregarding any information about types at run time. Execution time testing will be required for constraints only. Finally, static type checking allows the typechecker to give the correct meaning to overloaded operators, i.e; operators which can be used with operands of different types.

5. Class elements possess an abstract type and are the only values which can be destroyed. Predefined assertions on classes are provided and, if not otherwise specified, the operators for including or eliminating elements of a class are automatically defined.

6. A structured control structure is provided for failures and their handling.

The following definition of a simple schema illustrates Galileo. The example concerns departments and employees in a firm. The definitions are collected in the Organization schema.

```
Organization:=
  (rec Departments class
       Department ↔
         (Name: string
         and Budget: var num
         and Address: Address
         and Manager: var Employee
         and Employees: var seq Employee)
         key (Name)

  and Employees class
       Employee ↔
         (Name: string
         and Salary: var num
         and Dept: = Department
         key (Name)

  and NewEmployee (Name: string,
                   Salary: num,
                   NameOfDept: string
                   ) : Employee:=

       use ADept:=
             get Departments
                  with Name=NameofDept
             if-fails failwith "unknown dept."
       ext AnEmployee:=
             mkEmployee (Name:= Name
                         and Salary:= var Salary
                         and Dept:= ADept)
       in
         (Employees of ADept ←
            Employees of ADept append[ AnEmployee]
         AnEmployee)
```

```
and VipEmployee subset of Employees class
    VipEmployee ↔
       (is Employee
         and VipProperty: string)

and type Address:= (Street: string
                    and Zip: string
                    and City: string)
drop mkEmployee
)
```

The **rec** is used for recursive functions or for mutually dependent types, such as Department an Employee.

Departments an Employees are examples of base classes, while **key** in an example of predefined constraint to assert that the elements of the classes must differ in the value of the Name attribute.

An attribute can be modified if and only if it is defined of type **var**, otherwise it is constant and any attempt to update the value is detected statically.

The function NewEmployee is an example of a defined operation included in the schema. It is the only operation which can be used to create new elements of the class Employee since the **drop** operator prevents the predefined mkEmployee operation from being exported outside the schema definition. For Departments and VipEmployees the functions mkDepartment and mkVipEmployee are available.

VipEmployees is an example of a derived class. It contains all those employees who are believed to be very important. The elements of a derived class must have a type which is a subtype of the elements of the parent class. For instance, the type of the elements of VipEmployees is that of Employee with the additional attribute VipProperty.

This example shows how classes are used to deal with sets of interrelated objects. The approach has some similarity to that adopted for relational databases: In both cases the associations among data are described by means of the value of an attribute. However, in relational databases data are tuples of simple values, collected in relations, and the associations among them are represented by assigning as value to an attribute the key value of another tuple. In Galileo, instead, the mechanism of "data sharing" is used to represent associations, so that an element of a class can be shared as component by many others.

3. TYPE HIERARCHIES

An Important property of Galileo is the notion of subtype: if a type u is a subtype of a type v (u **is** v), then a value of the type u can be used in any context where a value of the type v is expected, but not vice versa, i.e. the subtype relation is a partial order. For instance, if a function f has a formal parameter of type v, then an application of f to a value of type u is correctly typechecked because no run time errors can occur. It is important to stress the point that, since Galileo has a secure type system, the notion of type hierarchies is related to that of well typed expression (Gordon 79a): Expressions which are syntactically well-typed are always semantically well-typed, i.e. the expressions do not cause run-time type errors and give a value of the correct type, if they terminate. In Milner's words "well-typed expressions do not go wrong" with hierarchies among types (Milner 78).

This notion of type hierarchies is different from the subtype concept of ADA, which is essentially a mechanism to give another name for a type whose set of values has been constrained, but is similar to the subclass machanism of Simula 67 (Birtwistle 73) and Smalltalk (Ingalls 78). The interesting aspects of the way it is used in Galileo is that this notion is extended to all the types, in the sense explained in the sequel, while preserving the important property that the language is still strongly typed.

With this mechanism Galileo supports the notion of programming by data specialization originally introduced by Simula 67 and generalized in TAXIS to all the constituents of a database application: Data, transactions, assertions and scripts (Borgida 82). Complex software applications, especially those employing databases, can be designed and implemented incrementally: Once a set of functions has been designed and tested for the most general data, they can still be used with data of any subtype introduced later on in the software development process. Moreover, new functions on the subtypes can be defined by the composition of the old functions with specific expressions.

The type system of Galileo includes primitive types and constructors to introduce user defined types, both concrete and abstract. For concrete types the type equivalence rule is the so-called structural equivalance: User-defined types names are just used as an abbreviation for the structure they represent. For abstract types the type equivalence rule is the so-called name equivalence rule: Two user-defined types are always different, and are different from the representation type.

User defined concrete types are tuples (record), sequences, discriminated unions (variants), modifiable values (references) and functions. For these types the subtype relation is automatically inferred by the typechecker according to the

following rules (Albano 82):

1. For any type t, (t **is** t).

2. If r and s are tuple types, of the form "(l1:t1 **and**.....**and** ln:tn)", then (r **is** s) iff:

   a. The set of labels of r contains the set of labels of s, and

   b. if r' and s' are the types of a common label, then (r' **is** s').

3. If r and s are variant types, of the form "< l_1 :t_1 **or** ....**or** ln:tn> ", then (r **is** s) iff:

   a. The set of labels of r is contained in the set of labels of s, and

   b. if r' and s' are the types of a common label, then (r' **is** s').

4. If r and s are sequence types, of the form "**seq** t", with elements of types r' and s' then (r **is** s)iff (r' **is** s').

5. If r and s are modifiable types, of the form "**var** t", then (r **is** s) iff the associated types are the same.

6. If (r → s) and (r' → s') are function types, then (r → s) **is** (r' → s') iff (r' **is** r), and (s **is** s').

For instance, if

```
type (Address:= (Street: string
                 and Zip: string
     and VipAddress:= (Street: string
                       and Zip: string
                       and Country: string)
     ext Person:= (Name: string
                   and Address: var Address)
     and Student:= (Name: string
                    and Address: var Address
                    and School: string
     and VipPerson:= (Name: string
                      and Address:var VipAddress))
```

then

Student **is** Person, and
VipAddress **is** Address

while it is false that

Person **is** VipPerson
Person **is** Student,
Student **is** VipPerson,
VipPerson **is** Person, and
VipPerson **is** Student

To define abstract types, Galileo provides two constructors. One, which will not be discussed here, is similar to CLU clusters, Alphard forms and Euclid modules. The other is similar to ADA types and will be presented by an exemple:
type Time ↔ (Hours: int **and** Minutes: int)

This declaration introduces:

1. The new type Time with a domain isomorphic to tuples.

2. the identifiers mkTime and repTime bound to two primitive functions, automatically declared, to map values of the representation type into the new one, and vice versa.

3. The selectors "Hours of" and "Minutes of", wich are primitive operators on the representation type. That is to say, primitive operators are inherited by the new type, with their names, but this overloading does not introduce ambiguities because the typechecker can infer the meaning of an operator from the type of the operands. A feature is also provided to restrict the set of operators to be inherited and to include assertions to be tested at run time (Albano 82).

For abstract types the subtype relation must be explicitly declared to the typechecker as follows:

Id **is** Id' ↔ t, where Id' ↔ t' and (t **is** t')

For instance:

```
type (Person↔ (Name: string
               and BirthDate: string
               and Address: string)

     ext Student is Person ↔
          (Name: string
          and BirthDate: string
          and School: string
          and Address: string))
```

The following abbreviation emphasizes the fact that the subtype Student inherits the attributes of the supertype Person:

```
type Student↔ (is Person
               and School: string)
```

Finally, multiple hierarchies are declared as Id **is** Id', Id" ↔ t, where (t **is** t') and (t **is** t") or in the abbreviated form "Id↔ **is** Id', Id", ...".

4. CLASSES

Classes are the mechanism to represent a data base by means of sets of modifiable interrelated objects. An element of a class is an object which is the computer representation of certain facts about an entity of the world that is being

modeled. An object-oriented view of a database is characterized by the following (Borgida 82, Kent 79, McLeod 82):

1. There is a one-to-one correspondence between objects in the database and entities of the world which are being modeled.
2. The objects of the database are all distinct and they might not have an external reference, such as a key, that stands for them.
3. Associations among entities are modeled by relating the corresponding objects and not external references. Moreover, only objects that exist in the database can be used to model associations.

A class is characterized by a name and the type of its elements. The name of a class denotes the elements of the class currently present in the data base, while the type gives the structure of the elements. The type of the class elements must be an abstract type; therefore two elements of different classes are always of different type, although they may be defined to have the same representation.

Elements of classes are the only values in Galileo which can be destroyed. Moreover, they are uniquely represented and when updated, their modification is reflected in all other objects in which they appear as components.

Each class can be either a **base class** or a **derived class.** A base class is defined independently of other classes, while a derived class is defined in terms of other classes. As in SDM (Hammer 81), a base class is used to model a primitive collection of entities, while a derived class is used to model alternative ways of looking at the same entities.

Base Classes

A base class is defined by the environment operator class, as shown in the following example with two mutually defined classes.

```
rec Departments class
        Department ↔
           (Name: string
           and Budget: var num
           and Address: string
           and Manager: var Employee
           and Employees:var seqEmployee)
           key (Name)
and Employees class
        Employee ↔
           (Name: string
           and Salary: var num
           and Dept: Department)
           key (Name)
```

The class operator introduces the following bindings:

1. The names Department and Employee bound to new types isomorphic to tuples.
2. The classes identifiers Departments and Employees bound to modifiable sequences of values of types Department and Employee.
3. The names mkDepartment and mkEmployee bound to two primitive functions, automatically declared, which differ from the similar functions on abstract types in that every time they are applied, new elements are created and are also automatically inserted into the associated classes, if the specified constraints are not violated. The constructed elements are also the values returned by the functions.
4. The functions repDepartment and repEmployee to map elements of the classes into the representation type.

The above declaration defines the structure of the objects together with a few constraints, some of which are predefined constraints to be tested when a class is modified:

a. The key constraints asserts that elements of a class must differ in the value of certain attributes. Note that if the key constraints is not specified, the insertion will be made even though the value of the attributes are equal to those of another object already present in the class. That is, elements of classes are always distinct objects, but the construction of an element will fail when the constraints are violated.

Other constraints are specified directly in the definition of element types:

b. Only attributes with a var type can be modified.

c. The attributes Employees and Manager in Departments are used to model the **part-of** relationship of Semantic Networks, which imply the followings dependency constraints: an employee cannot be eliminated from the database as long as he is a component of a department.

Derived Classes

In Galileo the two notions behind the **IS-A** hierarchy are expressed with two distinct mechanisms: The type hierarchy, to deal with the intensional aspect, and the derived class to deal

with the extensional aspect. A derived class implies an existency constraints among its elements and those of the parent class, i.e. the elements of a derived class are also elements of its parent class. The type of the elements of a derived class must be a subtype of the element type of the parent class. As a consequence of the subtype hierarchy, the elements of a derived class can be used as actual parameter for any operation defined for the elements of its parent class.

There are three ways of defining a derived class: by subset, partition or restriction /Albano 82/. Let us consider the first one which is the mechanism to model **IS-A** hierarchies.

A subset class contains a subset of the elements of the parent class which have been included explicitly with the proper operator. When a new element is added to a subset class, then it becomes also an element of the parent class.

Classes can also be derived from more than one parent class, with the restriction that the type of the element must be a subtype of all the element type of parent classes.

For example:

```
Secretaries subset of Employees class
  Secretary ↔
     (is Employee and Position : string)

FemaleEmployees subset of Employees class
  FemaleEmployee ↔
     (is Employee and Maternities : var num)

FemaleSecretaries subset of
     Secretaries, FemaleEmployees class
  FemaleSecretary ↔ (is Secretary,
                          FemaleEmployee)
```

The Employees are specialized in two overlapping subset classes, which in their turn are parents of another derived class.

5. CONCLUSIONS

The problem of integrating Semantic Data Models features in a strongly typed programming language has been addressed. A solution has been shown in the framework of the language Galileo, designed specifically with the above goal in mind. In particular, the generalization abstraction mechanism has been examined and it has been shown that to achieve a true integration of this feature in a strongly typed language, the type system should support the notion of type hierarchies.

The presentation has been informal, but this notion derive naturally from semantic considerations. The approach adopted is based on a previous result of Cardelli /82/: He has proved, in the framework of Edimburgh-ML, a semantic soundness theorem for a type system with multiple inheritance of types, based on Milner's theory of polymorphism. We are currently working on the proof of the theorem for the Galileo type system.

A preliminary implementation of a Galileo subset have been described in /Albano 83/. Presently, the final definition of the language has been completed and a more efficient implementation is in progress. This is being done by extending the ML implementation made by Cardelli on a VAX 11/780 running the UNIX(*) operating system.

The implementation of Galileo, for the time being, is for a single user environment and it does not include mechanisms for efficient recovery and concurrency control. In fact, the intended implementation is not to release a DBMS based on a Semantic Data Model, although ADAPLEX has shown that the time is mature for this kind of DBMS's too. Our main concerns are:

a. To test the features of the language for conceptual database design;

b. To study the architecture of a Database Designer's Workbench, the basic facilities, and tools to support the database design process (Albano 83).

ACKNOWLEDGEMENTS

I am indebted to Luca Cardelli and Renzo Orsini for their contribution to the design of Galileo. Also many thanks to Sol Greenspan, John Mylopoulos and the members of the Galileo Project for their constructive criticism to the contents of the paper.

REFERENCES

Albano A., L. Cardelli and R. Orsini /82/, "Galileo: A Strongly Typed, Interactive Conceptual Language", Technical Report, Department of Computer Science, University of Toronto (submitted for publication).

Albano A. and R. Orsini /83/, "Dialogo: An interactive Environment for Conceptual Design in Galileo", in **Methodology and Tools for Database Design**, S. Ceri (ed.), North-Holland, Amsterdam, 229-253, 1983.

(*) UNIX is a Trademark of Bell Laboratories.

Amble T., K. Bratberggensen and O. Risnes /79/, "ASTRAL, A Structured and Unified Approach to Database Design and Manipulation", in **Data Base Architecture**, G. Bracchi and G.M. Nijssen (eds), North-Holland, Amsterdam, 1979.

Biller, H. and E.J. Neuhold /78/, "Semantic of Databases: The Semantics of Data Models", **Information Systems 3**,1,11-30, 1978.

Birtwistle G.M., O-J Dahl, B. Myhrhang and K. Nygaard /73/, **"SIMULA Begin"**, New York, Petrocelli, 1973.

Borgida A.T., J. Mylopoulos and H.K.T. Wong /82/, "Methodological and Computer Aids for Interactive Information Systems Design", in **Automated Tools for Information System Design**, H.J. Schneider and A. Wasserman (eds), North-Holland, Amsterdam, 109-124, 1982.

Brodie M.L. /80/, "The Application of Data Types to Database Semantic Integrity", **Information System 5**, 4, 287-296, 1980.

Brodie M.L. and S.N. Zilles (eds) /81/, **Proc. Workshop on Data Abstraction, Data Bases and Conceptual Modelling, ACM SIGMOD Special Issue 11**, 2, 1981.

Capaccioli M. /83/, "La semantica Denotazionale del Galileo", Tesi di laurea in Scienze dell'informazione, Università di Pisa, Italy, 1983.

Carbonell J.G. /81/, "Default Reasoning and Inheritance Mechanism on Type Hierarchies", in **Proc. Workshop on Data Abstraction, Data Bases and Conceptual Modelling**, Brodie M.L. and S.N. Zilles (eds), **ACM SIGMOND Special Issue 11**, 2, 107-109, 1981.

Cardelli L. /82/, "Semantics and Typechecking of Multiple Inheritance" (draft).

Gordon M. /79a/, **"The Denotational Description of Programming Languages. An Introduction"**, Springer-Verlag, New York 1979.

Gordon M., R. Milner and C. Wadsworth /79b/, **"Edinburgh LCF"**, Lecture Notes in Computer Science, Vol. 78, Springer Verlag, 1979.

Hammer M. and B. Berkowitz /80/, "DIAL: A programming Language for Data Intensive Applications", **Proc. of ACM SIGMOD Conference**, 1980.

Hammer M. and McLeod /81/, "Database Description with SDM: A Semantic Database Model", **ACM TODS 6**, 3, 351-386, 1981.

Ingalls D.H. /78/, "The Smalltalk-76 Programming Systems: Design and Implementation", **Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages**, Tuscon, Arizona, 9-16, 1978.

Kent W. /79/, "Limitations of Record-Based Information Models", **ACM TODS 4**, 1, 107-131, 1979.

Lampson B.W., J.J. Horning, R.L. London, J.G. Mitchell and G.L. Popek /77/, "Report On The Programming Language Euclid", **ACM SIGPLAN Notices 12**,2, 1977.

Liskov B.H., A. Snyder, A. Atkinson and C. Schaffert /77/, "Abstraction Mechanisms in CLU", **CACM 20**, 8, 564-576, 1977.

McLeod D. and R. King /82/, "Semantic Database Models", in **Principle of Database Design**, S.B. Yao (ed.), Prentice Hall, 1982 (to appear).

Milner R. /78/, "A Theory of Type Polymorphism in Programming", **Journal of Computer and System Science 17**, 348-375, 1978.

Mylopoulos J., P.A. Bernstein and H.K.T. Wong /80/, "A language Facility for Designing Database-Intensive Applications", **ACM TODS 5**, 2, 185-207, 1980.

Rowe L.A. and K.A. Shoens /79/, "Data Abstraction, Views and Updates in RIGEL", **Proc. of ACM SIGMOD Conference**, Boston, Mass., 71-81, 1979.

Schmidt J.W. /78/, "Type Concepts for Database Definition", in **Database: Improving Usability and Responsiveness**, B. Schneidermann (ed.), Academic Press, 215-244, 1978.

Schmidt J.W. and M. Mall /80/, "Pascal/R Report", University of Hamburg, Fachbereich Informatik, Report N.66, January 1980.

Shopiro J.E. /79/, "A Programming Language for Relational Database", **ACM TODS 4**, 4, 493-517, 1979.

Shaw M., W.A. Wulf and R.L. London /77/, "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", **CACM 20**, 8, 553-564, 1977.

Shaw M. /80/, "The impact of Abstraction Concerns on Modern Programming Languages", **Proceedings of the IEEE,** Vol. 68, N.9, 1119-1130, 1980.

Shaw M. (ed.) /81/, **"ALPHARD: Form and Content"**, Springer Verlag, New York, 1981.

Smith J.M., S. Fox and T. Lancers /81/, "Reference Manual for ADAPLEX", Technical Report CCA-81-02, Computer Corporation of America, January 1981.

Wasserman A.I. /79/, "The Data Management Facilities of PLAIN", **Proc. of the ACM SIGMOD Conference**, Boston Mass., 60-70, 1979.

Weber H. /78/, "A Software Engineering View of Data Base Systems", **Proc. 4th Int. Conf. on VLDB**, Berlin, 36-51, 1978.

Wegner P. /80/, **"Programming with Ada: An Introduction by Means of Graduated Examples"**, Englewood Cliffs N.J., Prentice-Hall, 1980.

Wong H.K.T. and J. Mylopoulos /77/, "Two Views of Data Semantics: A Survey of Data Models in Artificial Intelligence and Database Management", **INFOR 15**,3, 344-382, 1977.